Period 1 : Learning goals

https://docs.google.com/document/d/1wkCSwyz-SFn6C0KtqVtUt9NiYGnTa81yUEHdaES8UxY/edit

Explain and Reflect:

Explain the differences between Java and JavaScript + node. Topics you could include:

- that Java is a compiled language and JavaScript a scripted language
- Java is both a language and a platform
- General differences in language features.
- Blocking vs. non-blocking

JavaScript : JavaScript is a lightweight programming language(“scripting language”) and used to make web pages interactive. It can insert dynamic text into HTML. JavaScript is also known as browser’s language. JavaScript(JS) is not similar or related to Java. Both the languages have a C like a syntax and are widely used in client-side and server-side Web applications, but there are few similarities only.

Java: Java is an object-oriented programming language and have virtual machine platform that allows you to create compiled programs that run on nearly every platform. Java promised, “Write Once, Run Anywhere”.

Picture showing some of the differences between java and javacript

Java	JavaScript
Java is strongly typed language and variable must be declare first to use in program.In Java the type of a variable is checked at compile-time.	JavaScript is weakly typed language and have more relaxed syntax and rules.
Java is an object oriented programming language.	JavaScript is an object based scripting language.
Java applications can run in any virtual machine(JVM) or browser.	JavaScript code used to run only in browser, but now it can run on server via Node.js.
Objects of Java are class based even we can't make any program in java without creating a class.	JavaScript Objects are prototype based.
Java program has file extension ".Java" and translates source code into bytecodes which is executed by JVM(Java Virtual Machine).	JavaScript file has file extension ".js" and it is interpreted but not compiled,every browser has the Javascript interpreter to execute JS code.
Java is a Standalone language.	contained within a web page and integrates with its HTML content.
Java program uses more memory.	JavaScript requires less memory therefore it is used in web pages.
Java has a thread based approach to concurrency.	Javascript has event based approach to concurrency.
Java supports multithreading.	Javascript doesn't support multi-threading.

Explain generally about node.js, when it "makes sense" and *npm*, and how it "fits" into the node echo system.

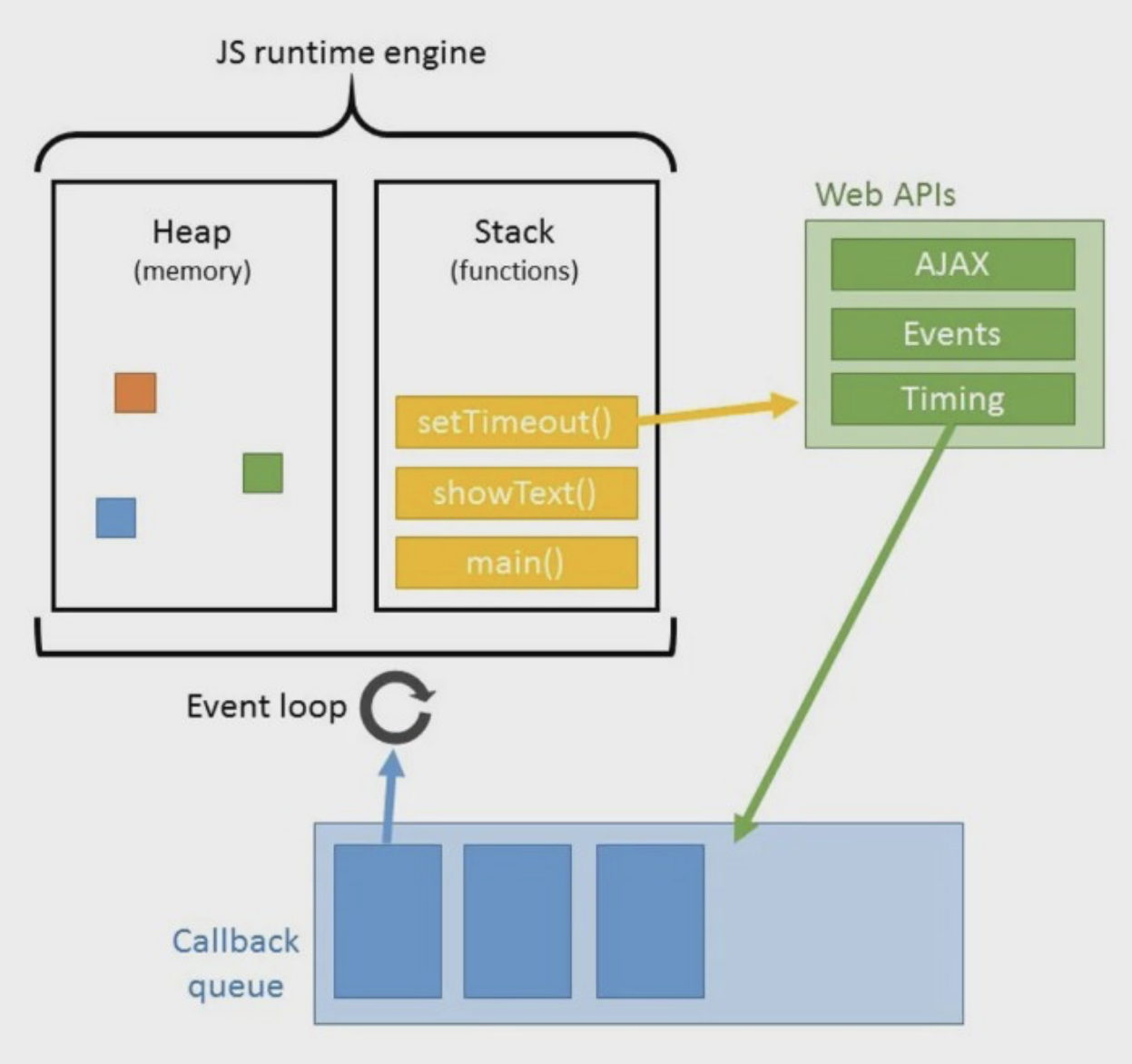
- Node is a way to run javascript code on your machine instead of in a browser
- NPM : Node package manager ⇒ A way to access different additional packages with features not native to JS. For an example "node-fetch".
- When using NPM a node_modules folder is created and to use packages in node_modules you have to import it

```
import fetch from "node-fetch";
```

Explain about the Event Loop in JavaScript, including terms like; blocking, non-blocking, event loop, callback queue and "other" API's. Make sure to include why this is relevant for us as developers.

- **Blocking** : When code is run synchronous, and therefore it needs to be executed before the program can keep on going
- **Non-blocking :** When code is run asynchronous and it therefore wait to execute code until there is time to run it / data has been loaded.
- **Call stack :** One thread = one call stack = one thing at a time. The call stack manages execution contexts and put tasks in a queue.

  JavaScript is singlethreaded at runtime, which means it can run one function at a time.
  The call stack records where we are in a program at runtime and if an error happens it will print a stacktrace (call stack).
- **Event loop** : looks at stack and task queue. If the stack is empty, it pushes the callback into the stack and the codeblock will be executed.
  Don't block the event loop → It will destroy the smoothness of a user interface.



Why is it relevant for us developers : It makes the code run way smoother when code is non-blocking and asynchronous since the program wont freeze waiting for the task to finish.

What does it mean if a method in nodes API's ends with xxxxxxSync?

- That it is a synchronous way to run the code.
- It is always better to run the methods asynchronous.
- If there is no Sync behind methods it is async since Node is async by defualt.

Explain the terms JavaScript Engine (name at least one) and JavaScript Runtime Environment (name at least two)

- A JavaScript engine is the program that converts and executes JS code into what we see and experience in browsers or Node
- **Chrome**: V8, **Mozilla**: SpiderMonkey, **Safari**: JavaScriptCore, **Internet Explorer**: Chakra.

Explain (some) of the purposes with the tools *Babel* and *WebPack and how they differ from each other*

Babel: A way to use new JS technologies in older JS version. Example ES6 features in ES5.

Webpack: Package manager to bundle JS code and define scripts, imports.

Webpack has a webpack.config.js file where every dependency can be defined

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

module.exports = {
  entry: {
    index: "./src/index.js",
    print: "./src/print.js",
  },
  devtool: "inline-source-map",
  devServer: {
    contentBase: "./dist",
  },
  plugins: [
    new CleanWebpackPlugin({ cleanStaleWebpackAssets: false }),
    new HtmlWebpackPlugin({
      title: "Output Management",
    }),
  ],
  output: {
    filename: "[name].bundle.js",
    path: path.resolve(__dirname, "dist"),
    publicPath: "/",
  },
  module: {
    rules: [
      {
```

```
        test: /\.css$/,
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: "asset/resource",
      },
    ],
  },
};
```

Explain using sufficient code examples the following features in JavaScript (and node)

Variable/function-Hoisting

- Hoisting: At løfte op
- Hoisting is that variables of the type var will be lifted up and declared prior to code executing. This means that in the following example we will get an undefined error. Since var 7 is hoisted

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

- Best practice is to declare variables at the top to avoid undefined errors.

Example: Check week1 folder

this in JavaScript and how it differs from what we know from Java/.net.

The JavaScript *this* keyword refers to the object it belongs to.

```
var person = {
  firstName: "John",
```

```
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

It has different values depending on where it is used:

- In a method, `this` refers to the **owner object**.

```
fullName : function() {
  return this.firstName + " " + this.lastName;
}
```

- Alone, `this` refers to the **global object**.

  When used alone, the **owner** is the Global object, so `this` refers to the Global object.

  In a browser window the Global object is `[object Window]`:

```
var x = this;
```

- In a function, `this` refers to the **global object**.

In a JavaScript function, the owner of the function is the **default** binding for `this`.

So, in a function, `this` refers to the Global object `[object Window]`.

```
function myFunction() {
  return this;
}
```

- In a function, in strict mode, `this` is `undefined`.

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, `this` is `undefined`.

```
"use strict";
function myFunction() {
```

```
  return this;
}
```

- In an event, `this` refers to the **element** that received the event.

```
<button onclick="this.style.display='none'">
  Click to Remove Me!
</button>
```

- Methods like `call()`, and `apply()` can refer `this` to **any object**.

In this example the person object is the "owner" of the function

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Example: Check week1 folder

Function Closures and the JavaScript Module Pattern

- This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

```
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();
```

- The variable `add` is assigned to the return value of a self-invoking function.

- The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

A closure is a special kind of object that combines two things:

- A function
- The environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created

Example: Check week1 folder

User-defined Callback Functions (writing functions that take a callback)

Explain the methods map, filter and reduce

Map : Creates a new array populated with the results of calling a provided function on every element in the calling array

```
var numbers = [65, 44, 12, 4];
var newarray = numbers.map(myFunction)
var example2 = numbers.map(number => number * 2)

function myFunction(num) {
  return num * 10;
}
```

Filter : Creates a new array with all elements that pass the test implemented by the provided function

```
var ages = [32, 33, 16, 40];

function checkAdult(age) {
  return age >= 18;
}

function myFunction() {
  var filteredAges = ages.filter(checkAdult);
}
```

Reduce : Reducer function (that you provide) on each element of the array, resulting in single output value

```
var numbers = [175, 50, 25];

var reducedNumbers =  numbers.reduce(myFunc);

function myFunc(total, num) {
  return total - num;
}
```

- Code example: Subtract the numbers in the array, starting from the left: Result is 100

Example: Check week1 folder

Provide examples of user-defined reusable modules implemented in Node.js (learnynode - 6)

```
var fs = require("fs");
var path = require("path");

module.exports = function (dirname, ext, callback) {
  var extension = "." + ext;
  fs.readdir(dirname, function (err, files) {
    if (err) {
      callback(err, null);
    } else {
      result = [];
      files.forEach(function (entry) {
        if (path.extname(entry) == extension) {
          result.push(entry);
        }
      });
      callback(null, result);
    }
  });
};
```

```
var lslib = require("./mymodule");

var dirname = process.argv[2];
```

```
var ext = process.argv[3];

lslib(dirname, ext, function (err, files) {
  for (i = 0; i < files.length; i++) {
    console.log(files[i]);
  }
});
```

Example: Check week1 folder

Provide examples and explain the es2015 features: let, arrow functions, this, rest parameters, destructuring objects and arrays, maps/sets etc.

See here for mere: https://babeljs.io/docs/en/learn/

Let & const : Block-scoped binding constructs. let is the new var. const is single-assignment. Static restrictions prevent use before assignment.

Arrow functions : Arrows are a function shorthand using the ⇒ syntax.

Unlike functions, arrows share the same lexical this as their surrounding code. If an arrow is inside another function, it shares the "arguments" variable of its parent function.

This : In JavaScript *this* keyword refers to the object it belongs to

- It has different values depending on where it is used
- In a method, this refers to the owner object.
- When used alone, the **owner** is the Global object, so this refers to the Global object.
- In a browser window the Global object is [object Window]:
- In Java, this refers to the current instance of an object which is used by a method

Rest parameters : The rest parameter syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic

functions in JavaScript.

```
function f(a, b, ...theArgs) {
  // ...
}
```

Destructuering objects and arrays : Destructuring allows binding using pattern matching, with support for matching arrays and objects.

```
const { name, realName } = hero;
```

Map : Efficient data structures for common algorithms

```
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;
```

Sets : Efficient data structures for common algorithms

```
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;
```

Provide an example of ES6 inheritance and reflect over the differences between Inheritance in Java and in ES6.

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance

```
class Person {
  constructor(first, last, age, gender, interests) {
    this.name = {
```

```
      first,
      last
    };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }

  greeting() {
    console.log(`Hi! I'm ${this.name.first}`);
  };

  farewell() {
    console.log(`${this.name.first} has left the building. Bye for now!`);
  };
}
```

```
class Teacher extends Person {
  constructor(subject, grade) {
    this.subject = subject;
    this.grade = grade;
  }
}
```

- In ES6 there isn't much of a difference between java and javascript classes.

  With ES6 classes came to JS and that made JS into a OOP language like Java

- You can inherit in ES6 like in java

Explain and demonstrate, how to implement event-based code, how to emit events and how to listen for such events

The code that is being imported with has created an eventlistener with the emit function.

```
const EventEmitter = require("events");

var url = "http://mylogger.io/log";

class Logger extends EventEmitter {
  log(message) {
    console.log(message);

    // Raise an event
    this.emit("messageLogged", { id: 1, url: "http://" });
  }
```

```
}

module.exports = Logger;
```

The code that is imported and called, the eventlistener is registrered with the on() method that takes a callback function:

```
const EventEmitter = require("events");

const Logger = require("./logger");
const logger = new Logger();

//register a listener
logger.on("messageLogged", (arg) => {
  console.log("Listener called", arg);
});
```

Example: Check week2 folder

ES6,7,8,ES-next and TypeScript

Provide examples with es-next running in a browser, using Babel and Webpack

Babel is a JS-transcompiler that can convert newer JS code to older versions.

See week 4 folder: react_babel_from_scratch

Explain the two strategies for improving JavaScript: Babel and ES6 + ES-Next, versus Typescript. What does it require to use these technologies: In our backend with Node and in (many different) Browsers

ES6, ES-next, JS, Babel:

- Discussed in meetings and it takes long time for new features to be added for the following stages:
  Proposal - Draft - Candidate - Finished
- The "big" guys on the market controls it
- JS is transcompiled using babel and webpack.
  → There is a barbell file in where you can setup your enviroment

```
{
  "presets": [
    ["env", {
      "targets: {
          "browsers": ["last 2 versions"]
        }
      }]
    ]
  ]
}
```

- As a user you will have to install alot of packages using NPM

  → example : npm install —save-dev babel-preset-es2015.

TypeScript

- Open source, using github, everyone can contribute
- Has to install TypeScript globally on machine or for bigger project in the project using NPM.
- TS has to be compiled to JS ⇒ You can configure a file that defines how / where the converted files should be saved.

Command : tsc —init ⇒ Creates the TS config file which has a lot of features. It is smart to define the output dir and set sourceMap to true for easier debugging.

Provide examples to demonstrate the benefits of using TypeScript, including, types, interfaces, classes and generics

Type defined variable

```
let myName: string = "Alice";
```

Class

```
class Point {
  x: number;
```

```
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

Interface

```
interface IMessage {
    original:string,
    upperCased:string
  }
```

Generic

```
class GenericLogger<T, U> {
  constructor() {}
  log = (a: T, b: U) => console.log(`Value 1: ${a}, value 2: ${b}`);
}
```

Putting it all together

```
interface IPerson {
  name: string;
}
interface IAddress {
  street: string;
}

class Person implements IPerson {
  //private _name : String;
  #name: string;
  constructor(name: string) {
    this.#name = name;
  }
  get name(): string {
    return this.#name;
  }
  set name(name: string) {
    this.#name = name;
  }
  toString(): string {
    return this.#name;
```

```
  }
}

class Address implements IAddress {
  //private _name : String;
  _street: string;
  constructor(street: string) {
    this._street = street;
  }
  get street(): string {
    return this._street;
  }
  set street(street: string) {
    this._street = street;
  }
  toString(): string {
    return this._street;
  }
}

function loggerV3(a: IPerson, b: IAddress) {
  console.log(`Person ${a.name}, adress: ${b.street}`);
}

let p1 = new Person("Kurt Wonnegut");
let a1 = new Address("Lyngby Hovedgade 23");

loggerV3(p1, a1);

class GenericLogger<T, U> {
  constructor() {}
  log = (a: T, b: U) => console.log(`Value 1: ${a}, value 2: ${b}`);
}

let personlogger = new GenericLogger<IPerson, IAddress>();
personlogger.log(p1, a1);
```

Example: Check week5 folder

Explain how we can get typescript code completion for external imports.

Installing the NPM package with the @type notation will make it work for TypeScript with code completion from the IDE.

```
npm install @type/lodash
```

Explain the ECMAScript Proposal Process for how new features are added to the language (the TC39 Process)

- The process of making changes in the ECMAScript specification is done by the TC39, and naturally called the TC39 process. This process is built from five stages, starting with stage zero. Any proposal for a change in the specification goes through these stages without exception, when the committee must approve the progress from one stage to the next one.
- The process was born due to the conduct of ECMAScript 2015 edition, also known as ES6, which was a pretty huge release lasting very long without delivery (actually almost 6 years). Therefore, as of ECMAScript 2017, the editions have become smaller and are delivered every year, containing all proposals which are accepted at all stages of the process, since the previous edition.
- The TC39 process have 4 stages - Proposal - Draft - Candidate - Finished
- Link : https://nitayneeman.com/posts/introducing-all-stages-of-the-tc39-process-in-ecmascript/

Callbacks, Promises and async/await

Explain about (ES-6) promises in JavaScript including, the problems they solve, a quick explanation of the Promise API and:

- JavaScript is single threaded, meaning that two bits of script cannot run at the same time; they have to run one after another. A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
- They solve the problem of only being able to run one thing at a time. A promise can be invoked and then later used, either rejected or resolved
- The Promise object works as a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
- This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method

returns a promise to supply the value at some point in the future.

- Promise API
  - Promise.resolve
    - The static Promise.resolve function returns a Promise that is resolved.
  - Promise.reject
    - The static Promise.reject function returns a Promise that is rejected.
  - Promise.all
    - The Promise.all() method takes an iterable of promises as an input, and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises.
  - Promise.race
    - The Promise.race() method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

Example(s) that demonstrate how to execute asynchronous (promise-based) code in serial or parallel

- **Check week 3 exericses**

Example(s) that demonstrate how to implement our own promise-solutions.

- **Check week 3 exericses**

Example(s) that demonstrate error handling with promises

- **Check week 3 exericses**

Explain about JavaScripts async/await, how it relates to promises and reasons to use it compared to the plain promise API.

- First, async/await makes the asynchronous code appear and behave like synchronous code. Just like Promises themselves, async/await is equally non-blocking. The purpose of async/await functions is to simplify the behavior of using Promises synchronously and to perform some behavior on a group of Promises.
- Good reference : https://blog.pusher.com/promises-async-await/

Provide examples to demonstrate

- Why this often is the preferred way of handling promises
- Error handling with async/await
- Serial or parallel execution with async/await.
    - **Check week 3 exercises**